The rump kernel: A tool for driver development and a toolkit for applications

Justin Cormack
justin@netbsd.org

Abstract

The NetBSD rump kernel is a way to run device drivers outside the kernel. Until recently the most common use was as a tool for running tests on NetBSD subsystems. In the last two years much more infrastructure has been built around it so that a much wider set of uses are possible. I cover some of these new uses in this paper, in particular using the rump kernel as a tool for driver development, and as a way to use it to run NetBSD applications in new environments.

What is the rump kernel?

What is an operating system made up of? The large proportion is a set of drivers that abstract the messy details of hardware away, and present a layered interface. This might be taking a particular network card, and presenting a socket interface supporting TCP and UDP, or taking a SCSI hard drive and presenting a Posix file system interface. What the rump kernel does is provide these drivers outside the context of a traditional operating system, so they can be used as a library by applications. What it does not do is as important: unlike say a userspace kernel (like User Mode Linux[2] or NetBSD/usermode[8], there are parts it does not provide. These are memory allocation, threads and a scheduler. These must be provided by the platform the rump kernel application runs on.

A simple example of a platform is simply a userspace program. These already provide memory allocation (eg `posix_memalign`) and a thread implementation (eg pthreads) which the underlying operating system will schedule. An implementation that does this ships with NetBSD, allowing the NetBSD drivers to be run in NetBSD userspace. This is for example used for running tests on NetBSD subsystems, as for example multiple instances can be set up very fast to test sockets.

As the rump kernel just depends on a very small hypercall layer, just to provide the basics of memory allocation, threads and clocks, and because the underlying NetBSD code is very portable, the rump kernel hypercall layer for NetBSD can also be built for other Posix environments, and it is also possible to write a version for other environments such as running directly on Xen, or on bare metal, or within a different operating system, such as a microkernel operating system.

The rump kernel development was started by Antti Kantee as his PhD thesis work[1], completed in 2012. It was initially released as part of NetBSD 5.0 in 2009. Development has continued in the NetBSD tree and in external open source repositories[3] with a larger community of contributors.

Testing and debugging

The first use of the rump kernel was in the NetBSD tree for improving testing. NetBSD has an extensive test suite[9]. For many tests, it is easier to use a rump kernel rather than the host system as there are no side effects, so for example it is possible to make tests involving global network settings without worrying about interfering with the host machine, while being much faster than setting up a full virtual emulated system. The rump kernel works across all the platforms NetBSD supports, and starts up in 10ms, so testing is very fast. Because it is running exactly the same code as the underlying NetBSD system, compiled at the same time as the kernel being tested, it will have the same bugs, so the tests should find these.

As well as tests designed to run on a NetBSD system, tests can also be run on any other system that supports the rump kernel. For example the ljsyscall project[10] which I wrote provides a system call framework for LuaJIT[11] for NetBSD, Linux, FreeBSD and other platforms, and also supports calling the rump kernel instead of the underlying OS. This has an extensive test suite and has been used to find bugs in NetBSD. This runs extremely fast, so can be run multiple times to find race conditions that

would be hard to find under normal operating systems testing. This is also used for continuous integration of the rump kernel on the development version of NetBSD to pick up bugs as they are committed.

Another advantage of running a single application which has the userspace and kernel parts linked into a single binary is that normal userspace debugging tools can be used across both parts, This includes tools such as valgrind[12] as well as debuggers, and other static analysis and fuzzing tools. There is a lot of scope for further work in this area which is currently being explored.

When developing device drivers, one of the biggest issues is that they are liable to crash, so developing them in the kernel that you are working in is inconvenient. This can be alleviated by using a separate development machine or virtual machine, but this still has a slow modify, compile, reboot cycle.

With a rump kernel you can develop many drivers purely in userspace, even for physical hardware. For developing file system code, you can use a block device as the "physical device" backing the file system, while for the network stack you can use a standard Unix tap device which is a virtual ethernet device that feeds frames to and from the host stack.

For PCI devices there is currently a userspace PCI driver for Linux, currently using the Linux-specific uio interface. This works for many but not all PCI devices. Further drivers for other platforms that support userspace PCI access are planned, such as a Linux vfio driver, which supports iommu operations and a larger variety of devices, such as MSI interrupts.

The first major PCI driver developed was the Intel Centrino 7260 driver developed for NetBSD and OpenBSD by Antti Kantee. The commit message said "This is probably the world's first Canadian cross device driver: it was created for OpenBSD by writing and porting a NetBSD driver which was developed in a rump kernel in Linux userspace."[13]

Running Application code

Originally when programs were written to use the rump kernel they had to be explicitly written this way, using the rump kernel namespace. This was fine for tests, where it was explicit, and for frameworks such as ljsyscall where the framework can abstract this for the user so their Lua code does not see whether the rump kernel or the user's kernel is being used. However writing code specifically for a rump kernel is a lot of work, and it is against the philosophy of reusing as much existing code as possible.

The first step towards this was to build a version of NetBSD's libc against the rump kernel calls rather than the standard assembly system calls. This was first done on the Xen platform, at which point it was possible to compile applications like Lua which only use libc against a set of NetBSD headers and then link them to the rump kernel version of libc.

This was then upstreamed into NetBSD as a build option, and a framework to build the other libraries and applications that ship as part of NetBSD base was developed. These are also very useful as they include the basic commands to configure more complex kernel features, such as wireless cards, software RAID and encrypted block devices. This enables testing and development of these kernel subsystems using the standard commands.

As part of this work the rumprun-posix tools were built[7], that enabled the NetBSD libraries and tools to be built in Posix userspace, which is particularly useful for development. The problem here is the difficult mix of having both the host C library and the rump-targeted NetBSD C library linked into the same binary, as they share the same namespace. This was done by a link script that carefully renames symbols into a new namespace where necessary.

The rumprun-posix tools also allow the user to run a remote rump kernel, where the system calls are transported over a socket. Obviously this mode is not as performant is linking the rump kernel code into the binary, but it can be very useful. In particular it allows binaries to execute against a persistent long-running rump kernel, just like the normal execution model of kernel code. This means that for example a sequence of configuration commands can be given, then a test program can be run with the correct configuration.

Microkernels and unikernels

The rump kernel has been used as a way to supply device drivers in other new operating systems, which do not yet have a full set of device drivers. For example, Genode[4] is a framework for building microkernel operating systems using the L4 family of microkernels. Genode uses the rump kernel to provide file system support, so that it does not have to develop its own file systems.

Another use for the rump kernel is as a framework for building applications that include not just the application but also the just enough operating system that is required packaged together. This "unikernel" or "library kernel" concept has been developed before largely for specific languages such as Mirage[5] for OCaml, where the majority of the drivers are written from scratch, including the IP stack for example. While this has many advantages, such as allowing the development of typesafe libraries for critical user-facing functions, it also means that development is slower as existing drivers cannot be used. What the rump kernel offers is a set of unmodified, maintained, high quality drivers from NetBSD, which can be used for this type of development, perhaps in con-

junction with newly written drivers.

Current work has standard application components such as the Nginx web server[6], and languages such as PHP and Lua supported running directly under the rump kernel under Xen. As further build and integration issues are fixed, many other languages and existing programs should also run. This allows isolated applications to run in a microservice architecture, each with only its own dependencies, but without requiring a full host operating environment. This reduces attack surface significantly by removing components that are not required to run the application.

Future developments

Increased portability of the rump kernel is still an aim. While the NetBSD core provides excellent portability to different processor architectures, such as Powerpc, ARM and MIPS, there is still more portability work to do. There exists baremetal support on 32 bit x86, but not yet for 64 bit. There are plans to support bare metal ARM based devices for example, including microcontrollers that are unable to run a full operating system as they have no MMU as well as small machines such as the Raspberry Pi.

As mentioned earlier there are plans to increase the range of PCI devices that can be addressed by the rump kernel, and this may well be completed in time for AsiaBSDCon. A FreeBSD port is also feasible as FreeBSD has added userspace PCI driver support for Bhyve.

Most of the focus is on easier build and configuration to run applications, with the aim being to make building rump kernel targeted applications just as easy as native ones, just by changing the compiler in most cases. This means that more language frameworks need to be compiled, such as the C++ libraries which are in progress at present. Many languages and libraries should present no trouble, but others make certain assumptions about the way they are linked or run, or call system calls directly via assembly rather than through libc, so changes are needed.

The rump kernel work takes place both within the upstream NetBSD tree as well as in an active external community[3] which includes both NetBSD developers and users who primarily use other operating systems or environments. All code is BSD licensed and freely available.

Conclusion

NetBSD's rump kernel is an unusual feature that no other operating system has. However it is friendly and can be used to enhance other operating systems too. Whether your interests are in better testing, developing new code, or exploring new ways to run applications it is worth investigating.

References

[1] Antti Kantee, *Flexible operating system internals: the design and implementation of the anykernel and rump kernels*. Doctoral dissertation, Aalto University 2012.

[2] User Mode Linux, `http://user-mode-linux.sourceforge.net/`.

[3] Rump Kernels, `http://rumpkernel.org/`.

[4] Genode, `http://genode.org/`.

[5] Open Mirage, `http://www.openmirage.org/`.

[6] Nginx, `http://nginx.org/`.

[7] Rumprun Posix, `https://github.com/rumpkernel/rumprun-posix`.

[8] Reinoud Zandijk, *NetBSD/usermode*. `http://www.13thmonkey.org/documentation/NetBSD/EuroBSD2012-NetBSD_usermode-paper.pdf` EuroBSDCon 2012.

[9] Antti Kantee, *Testing NetBSD: easy does it*. `http://blog.netbsd.org/tnf/entry/testing_netbsd_easy_does_it` 2010.

[10] Justin Cormack, ljsyscall, `https://github.com/justincormack/ljsyscall`.

[11] LuaJIT, `http://luajit.org/luajit.html`.

[12] Valgrind, `http://valgrind.org/`.

[13] Antti Kantee, `http://mail-index.netbsd.org/source-changes/2015/02/07/msg062979.html`.